Load Balancing in Distributed Applications Based on Extremal Optimization

Ivanoe de Falco, Eryk Laskowski, Richard Olejnik, Umberto Scafuri, Ernesto Tarantino, Marek Tudruj

Load Balancing in Distributed Applications Based on Extremal Optimization

I. De Falco[1], E. Laskowski[2], R. Olejnik[3], U. Scafuri[1], E. Tarantino[1], M. Tudruj[2,4]

[1]Institute of High Performance Computing and Networking, CNR, Naples, Italy
[2]Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
[3]Computer Science Laboratory, University of Science and Technology of Lille, France
[4]Polish-Japanese Institute of Information Technology, Warsaw, Poland
{laskowsk,tudruj}@ipipan.waw.pl, richard.olejnik@lifl.fr,
{ivanoe.defalco,umberto.scafuri,ernesto.tarantino}@na.icar.cnr.it

Abstract. The paper shows how to use Extremal Optimization in load balancing of distributed applications executed in clusters of multicore processors interconnected by a message passing network. Composed of iterative optimization phases which improve program task placement on processors, the proposed load balancing method discovers dynamically the candidates for migration with the use of an Extremal Optimization algorithm and a special quality model which takes into account the computation and communication parameters of the constituent parallel tasks. Assessed by experiments with simulated load balancing of distributed program graphs, a comparison of the proposed Extremal Optimization approach against a deterministic approach based on a similar load balancing theoretical model is provided.

Keywords: distributed program design, extremal optimization, load balancing;

1 Introduction

Efficient execution of irregular distributed applications usually requires some kind of processor load balancing. This is even more true in a multi-user environment where, following variations in system resources availability and/or changes of their computational load, the balance in using the executive resources can notably vary over time. Thus, a dynamic load balancing facility, embedded in the runtime environment or in the distributed application is essential.

The contribution of this paper is a dynamic load balancing method based on the program and runtime system behavior observation supported by the Extremal Optimization (EO) approach [1]. The presented approach leverages some our earlier works on load balancing, reported in [2,3], and our experience in using Extremal Optimization to static task scheduling in distributed programs [4, 5]. The proposed load balancing algorithm is composed of iterative optimization phases which improve program task placement on processors to determine the

possibly best balance of computational loads and to define periodic migration of tasks. The Extremal Optimization is used in iterative load balancing phases which are executed in the background, in parallel with the application program. The Extremal Optimization algorithm discovers the candidate tasks for migration based on a special quality model including the computation and communication parameters of parallel tasks. The algorithm is assessed by experiments with simulated load balancing of distributed program graphs. In particular, the experiments compare the proposed load balancing method including the Extremal Optimization with an equivalent deterministic algorithm based on the similar theoretical foundations. The comparison shows that the quality of load balancing with Extremal Optimization is in most cases better than that of the deterministic algorithm.

A good review and classification of load balancing methods is presented in [6]. When we compare the proposed load balancing method to the current parallel computing environments with load balancing like CHARM++ [7] or Zoltan [8], we notice that none of them includes Extremal Optimization as a load balancing algorithm component. So, the proposed approach has clear originality features and enables making profit of the Extremal Optimization advantages such as low computational complexity and limited use of memory space.

The paper consists of three parts. In the first part the Extremal Optimization principles are shortly explained. The second part describes the theoretical basement for the discussed algorithm and explains how the Extremal Optimization is applied to the proposed method of dynamic load balancing. The third part explains the performed experimental assessment of the proposed algorithms.

2 Extremal Optimization algorithm principles

Extremal Optimization is an important nature inspired optimization method. It was proposed by Boettcher and Percus [1]. It represents a method for NP–hard combinatorial and physical optimization problems.

EO works with a single solution S consisting of a given number of components s_i, each of which is a variable of the problem and is thought to be a species of the ecosystem. Once a suitable representation is chosen, by assuming a predetermined interaction among these variables, a local fitness value ϕ_i is assigned to each of them. Then, at each time step the global fitness $\Phi(S)$ is computed and S is evolved, by randomly updating only the worst variable, to a solution S' belonging to its neighborhood $Neigh(S)$. An obtained solution is registered if its global fitness function is better than that of the best solution found so far.

To avoid sticking in a local optimum, we use a probabilistic version of EO based on a parameter τ , i.e., τ–EO, introduced by Boettcher and Percus. According to it, for a minimization problem, the species are first ranked in increasing order of local fitness values, i.e., a permutation π of the variable labels i is found such that: $\phi_\pi(1) \leq \phi_\pi(2) \leq \ldots \phi_\pi(n)$, where n is the number of species. The worst species s_j is of rank 1, i.e., $j = \pi(1)$, while the best one is of rank n. Then, a distribution probability over the ranks k is considered as follows: $p_k \sim k^{-\tau}$,

$1 \leq k \leq n$ for a given value of the parameter τ. Finally, at each update, a generic rank k is selected according to p_k so that the species s_i with $i = \pi(k)$ randomly changes its state and the solution moves to a neighboring one, $S' \in \mathit{Neigh}(S)$, unconditionally. The only parameters are the maximum number of iterations $\mathcal{N}_{\text{iter}}$ and the probabilistic selection parameter τ. For minimization problems τ–EO proceeds as in the Algorithm 1.

3 Extremal Optimization applied to load balancing

3.1 System and program model

An *executive system* consists of N computing nodes interconnected by a message passing network (e.g. a cluster of workstations). Each node, identified by an integer value in the range $[0, N-1]$, is a multicore processor.

Distributed application programs are composed of processes and threads inside each process. The application model is similar to the model presented in [9] (Temporal Flow Graph, TFG). The application consists of T indivisible tasks (these are threads in processes). Each task consists of several computational instruction blocks, separated by communication with other tasks.

The *target problem* is defined as follows: assign each task $t_k, k \in \{1 \ldots |T|\}$ of the program to a computational node $n, n \in [0, N-1]$ in such a way that the total program execution time is minimized, assuming the program and system representation as described earlier in this section.

The load balancing approach, proposed in the paper, consists of two main steps: the **detection** of the imbalance and its **correction**. The first step uses some measurement infrastructure to detect the functional state of the computing system and executed application. In parallel with the execution of an application, computing nodes periodically report their loads to the load balancing controller which evaluates the current system load imbalance value. Depending on this value the second step (i.e. the imbalance correction) is undertaken or the step

Algorithm 1 General EO algorithm

initialize configuration S at will
$S_{\text{best}} \leftarrow S$
while maximum number of iterations $\mathcal{N}_{\text{iter}}$ not reached **do**
  evaluate ϕ_i for each variable s_i of the current solution S
  rank the variables s_i based on their fitness ϕ_i
  choose the rank k according to $k^{-\tau}$ so that the variable s_j with $j = \pi(k)$ is selected
  choose $S' \in \mathit{Neigh}(S)$ such that s_j must change
  accept $S \leftarrow S'$ unconditionally
  if $\Phi(S) < \Phi(S_{\text{best}})$ **then**
    $S_{\text{best}} \leftarrow S$
  end if
end while
return S_{best} and $\Phi(S_{\text{best}})$

one is repeated. In the second step, we execute the EO-based load balancing algorithm described in next sections, which determines the set of tasks for migration and the migration target nodes. Then we perform the physical task migrations following the best solution found by the EO algorithm. Next we proceed to step one.

The *state of the system* is expressed in the terms of:
$Ind_{power}(n)$ — computing power of a node n, which is the sum of computing powers of all cores on the node,
$Time^{\%}_{\mathrm{CPU}}(n)$ — the percentage of the CPU power available for computing threads on the node n, periodically estimated by observation agents on computing nodes. The state metrics are used to detect and correct a load imbalance between nodes.

3.2 Detection of load imbalance

Computing nodes composing the parallel system can be heterogeneous, therefore to detect the current load imbalance in the system, we will base it on the percentage of the CPU power available $Time^{\%}_{\mathrm{CPU}}(n)$ for computing threads on the node n.

A current load imbalance LI is defined as the difference of the CPU availability between the most heavily and the least heavily loaded computing nodes composing the cluster, which can be determined as:

$$LI = \max_{n \in N}(Time^{\%}_{\mathrm{CPU}}(n)) - \min_{n \in N}(Time^{\%}_{\mathrm{CPU}}(n)) \geq \alpha$$

where: N — the set of all computing nodes. The current load imbalance requires a load balancing correction if LI is at least so big as a positive constant α. The value of the α is set using a statistical or/and experimental approach. Following our previous research [2] on load balancing algorithms for Java-based distributed environment, we can restrict the value to the interval $[0.25 \ldots 0.75]$.

Then, according to the load of each node, the set of computing nodes is divided into three categories, based on the computed power availability indexes: overloaded, normally loaded and underloaded. To build categories, we use the K-Means algorithm [10] with $K = 3$. The three centers that we choose are the minimum, average and maximum availability indexes, where the average index is simply the average of indexes measured during the last series of measures over the whole cluster.

3.3 Load balancing procedure

Now, we are able to perform the second step of our approach — migration of application's task to balance the load of the system.

The **state of the application** is characterized by two application-specific metrics, which should be provided by an application programmer:

1. $\mathrm{COM}(t_s, t_d)$ is the communication metrics between tasks t_s and t_d,
2. $\mathrm{WP}(t)$ is the load weight metrics of a task t.

A **mapping solution** S is represented by a vector $\mu = (\mu_1, \ldots, \mu_{|T|})$ of $|T|$ integers ranging in the interval $[0, N-1]$, where the value $\mu_i = j$ means that the solution S under consideration maps the i–th task t_i of the application onto computing node j. The number of processor cores is not represented inside the solution encoding, however, it is implicitly taken into account (via the $Ind_{power}(n)$ function) when estimating the global and local fitness functions while solving the scheduling problem.

The **global fitness** function $\Phi(S)$ is defined as follows.

$$\Phi(S) = attrExtTotal(S) * \Delta_1 + migration(S) * \Delta_2 + imbalance(S) * [1 - (\Delta_1 + \Delta_2)]$$

where $1 > \Delta_1 \geq 0$, $1 > \Delta_2 \geq 0$ and $\Delta_1 + \Delta_2 < 1$ hold.

The function $attrExtTotal(S)$ represents the total external communication between tasks for given mapping S. By "external" we mean the communication between tasks placed on different nodes (i.e. which have to be transmitted actually through communication links between computing nodes). The value of this function is normalized in the range $[0, 1]$, i.e. it is a quotient of an absolute value of the total external communication volume and the total communication volume of all communications (when all tasks are placed on the same node $attrExtTotal(S) = 0$, when tasks are placed in the way that all communication became external $attrExtTotal(S) = 1$):

$$attrExtTotal(S) = \frac{totalExt(S)}{\overline{\mathrm{COM}}}$$

where: $\overline{\mathrm{COM}} = \sum_{s,d \in T} \mathrm{COM}(s,d)$ and $totalExt(S) = \sum_{s,d \in T : \mu_s \neq \mu_d} \mathrm{COM}(s,d)$.

The function $migration(S)$ is a migration costs metrics. The value of this function is normalized in the range $[0, 1]$ by dividing by the total number of tasks (i.e. when all tasks have to be migrated $migration(S) = 1$, otherwise $0 \leq migration(S) < 1$).

$$migration(S) = \frac{|\{t \in T : \mu_t^S \neq \mu_t^{S*}\}|}{|T|}$$

where: S is the currently considered solution and $S*$ is the previous solution (or the initial solution at the start of the algorithm).

The function $imbalance(S)$ represents the load imbalance metrics in the solution S. It is equal to 1 when in S there exists at least one unloaded (empty) computing node, otherwise it is equal to the normalized average absolute load deviation in S.

$$imbalance(S) = \begin{cases} 1 & \text{exists at least one unloaded node} \\ D^*(S)/2 * N * \overline{\mathrm{WP}} & \text{otherwise} \end{cases}$$

where: $D^*(S) = \sum_{n \in [0,N-1]} |\mathrm{NWP}(S,n)/Ind_{power}(n) - \overline{\mathrm{WP}}|$,
$\mathrm{NWP}(S,n) = \sum_{t \in T : \mu_t = n} \mathrm{WP}(t)$, $\overline{\mathrm{WP}} = \sum_{t \in T} \mathrm{WP}(t) / \sum_{n \in [0,N-1]} Ind_{power}(n)$.

In the applied EO the **local fitness** function of a task $\phi(t)$ is designed in such a way that it forces moving tasks away from overloaded nodes, at the

same time preserving low external (inter-node) communication. The γ parameter ($0 < \gamma < 1$) allows tuning the weight of load metrics.

$$\phi(t) = \gamma * load(\mu_t) + (1 - \gamma) * rank(t)$$

The function $load(n)$ indicates whether the node n, which executes t is overloaded (i.e. it indicates how much its load exceeds the average load of all nodes):

$$load(n) = \frac{load^*(n)}{\max_{m \in [0,N-1]} load^*(m)}, \quad load^*(n) = \max(\frac{\mathrm{NWP}(S,n)}{Ind_{power}(n)} - \overline{\mathrm{WP}}, 0).$$

The $rank(t)$ function governs the selection of best candidates for migration. The migrated ones are those which have lower communication with their current node and which have the average load (in order not to change load balance too much nor too little):

$$rank(t) = 1 - (\beta * attr(t) + (1 - \beta) * ldev(t))$$

where: β is a real number between 0 and 1 — a parameter indicating the importance of the weight of attraction metrics. The comparison formulae are:
(1) the attraction of the task t to the actual computing node:

$$attr(t) = \frac{attr^*(t)}{\max_{o \in L(t)}(attr^*(o))}$$

where: $attr^*(t) = \sum_{o \in L^*(t)}(\mathrm{COM}(t,o) + \mathrm{COM}(o,t))$ — the amount of communication between task t and other tasks on the same node,
$L(t) = \{s \in T : \mu_t = \mu_s\}$ — the set of threads, placed on the same node as the thread t (including t),
$L^*(t) = \{s \in T, s \neq t : \mu_t = \mu_s\}$ — the set of threads, placed on the same node as a thread t (excluding t).
The formula above allows computing the attraction of a task to the local node in order to compare it with the attractions of other tasks on this node.
(2) the load deviation compared to the average quantity of work:

$$ldev(t) = \frac{ldev^*(t)}{\max_{o \in L(t)}(ldev^*(o))}$$

where: $ldev^*(t) = |\mathrm{WP}(t) - mean_{\mathrm{WP}}(t)|$, $mean_{\mathrm{WP}}(t) = \sum_{o \in L(t)} \mathrm{WP}(o)/|L(t)|$.

3.4 Deterministic approach for load balancing

For comparison purposes, we have implemented also a fully deterministic load balancing algorithm, based on similar migration criteria as shown in the previous section (see [3] for the description). Similarly to the EO approach, the load balancing triggering is controlled by the imbalance metrics LI. The deterministic algorithm iterates over all overloaded nodes and migrates a single task from each such node to an underloaded one. The task for migration inside an underloaded node is selected according to the value of the $rank(t)$ function. The target of migration is the node that minimizes the weighted sum of $attrExtTotal(S)$ and $Time^{\%}_{\mathrm{CPU}}(n)$.

4 Experimental assessment of load balancing algorithms

We will present now an experimental assessment of the presented load balancing algorithm. The experimental results have been obtained by simulated execution of application programs in a distributed system. The assumed simulated model of execution corresponds to parallelization based on message-passing, using the MPI library for communication. The applied simulator was built following the DEVS discrete event system approach [11].

Applications were run in a cluster of multi-core processors, each of which had its own main memory and a network interface. Communication contention was modeled at the level of the network interface of each computing node.

During experiments we used a set of 10 synthetic exemplary programs, modeled as TFG (see section 3.1). These programs were randomly generated, but their general structure resembled typical MPI-based parallel applications which correspond to numerical analysis or physical phenomena simulation. Each application program consisted of a set of program modules, Fig 1. A module was composed of parallel tasks (threads). Tasks of the same module communicated, at the boundaries between modules, there was a global exchange of data.

The number of tasks in an application varies from 16 to 80. The communication/computation ratio (the quotient of the communication time to the execution time in our experimental environment) for applications is in the range 0.05...0.15. Three applications have regular tasks' execution times. The difference between regular and irregular applications is that the execution time of tasks in some (or all) modules of the irregular applications depends on the processed data. From the external view, the irregular applications exhibit the behavior in which the execution time of tasks and the communication scheme seem unpredictable. Thus, in irregular applications, a load imbalance will occur in computing nodes in a natural way. In regular ones a load imbalance can appear due to the placement of multiple tasks on the same processor.

During experiments, we have used the following parameters: $\alpha = 0.5, \beta = 0.5, \gamma = 0.5, \Delta_1 = 0.25, \Delta_2 = 0.25, \tau = 1.5$. The number of iterations of the EO algorithm was set to 500. We simulated execution of applications in systems with

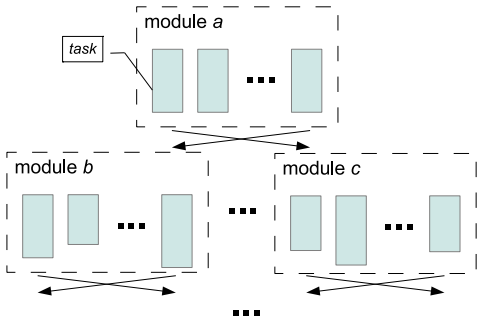


Fig. 1. The general structure of exemplary applications.

algorithm		2	3	4	8	average
EO – extremal optimization	irregular	18.66%	33.42%	37.32%	41.91%	35.08%
	regular	21.95%	31.85%	46.85%	34.21%	34.71%
DT – deterministic	irregular	18.27%	28.17%	33.97%	43.82%	33.80%
	regular	22.95%	29.54%	41.62%	35.02%	33.39%

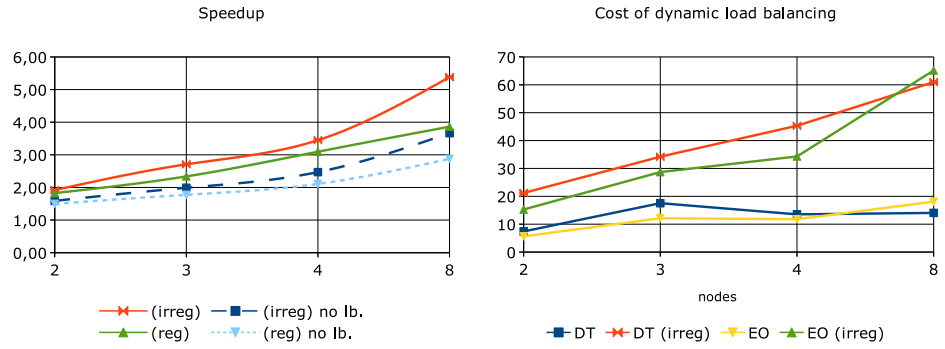
Fig. 2. Speed-up improvement for irregular and regular applications due to load balancing for different number of nodes in the system.

2, 3, 4 and 8 identical computing nodes. The results are the averages of 5 runs of each application, each run for 4 different methods of initial task placements (random, round-robin, METIS, packed) i.e. 20 runs for each parameter set.

The speed-up improvement resulting from load balancing performed by the EO-based algorithm and the deterministic approach (DT) is shown in Fig. 2. The general speed-up improvement over the execution without load balancing is bigger for EO-based algorithm. As we'll show later in this section, it is possible to obtain even better speedup by EO through proper parameter tuning.

In Fig. 3(a) the speed-up of irregular and regular applications for different number of computing nodes is shown. Our exemplary regular applications give smaller speed-up than irregular ones (with or without load balancing).

Since migration costs can be very different (the single migration can be as short as a simple task activation message, but also it can involve a transfer of the processed data, which is usually very costly), we decided to measure also the imposed load balancing costs as the number of tasks migrations. As shown in Fig. 3(b), the average cost imposed by EO algorithm is generally lower than the cost introduced by the deterministic approach.



(a) (b)

Fig. 3. (a) Speedup for different number of nodes with and without load balancing (b) Cost of the dynamic load balancing as the number of task migrations per single execution of an application.

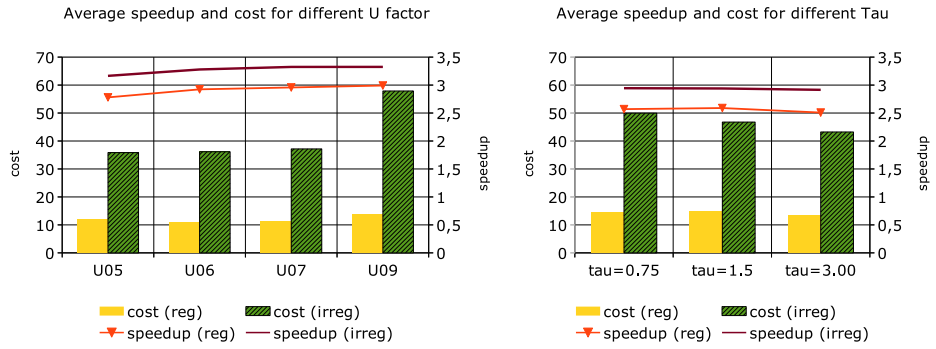
The big advantage of the EO approach is the ability to tune its behavior through the algorithm parameters setting. Thus, we can have bigger speedup at the higher cost of migration, or lower speedup together with a reduced migrations number. To do so, we performed the simulations for different values of $\tau \in \{0.75, 1.5, 3.0\}$ and a varying *Imbalance factor*, see table below. The *Imbalance factor* is a set of $\gamma, \Delta_1, \Delta_2$ values, which regulates the importance of the load imbalance in local and global evaluation functions of the EO algorithm.

Imbalance factor	γ	Δ_1	Δ_2
U05 (imbalance least important)	0.50	0.25	0.25
U06	0.60	0.18	0.22
U07	0.75	0.13	0.17
U09 (imbalance most important)	0.95	0.05	0.05

For the tested applications, better results were obtained when the load imbalance was the primary optimization factor, namely for U07 and U09, Fig. 4(a). On the other hand, for U09 the cost of load balancing was very high (the algorithm migrates tasks very often to maintain a perfect load balance). We expect that for applications with more intense communication (i.e. higher value of communication/computation ratio) better results can be obtained for U05 and U06. It will be investigated in further research.

Fig. 4(b) shows that an increasing value of τ decreases the number of migrations, it reduces also slightly the obtained speedup (note that increasing τ increases the probability of selection of the worst species for change).

Our experimental results collected so far by simulation confirm that the presented EO-based load balancing method performs well for different run-time conditions. Other important features of the method are the low migration overhead as well as the ease of programming and tuning the load balancing algorithm.



(a) (b)

Fig. 4. Speedup and cost for different values of EO parameters as a function of: (a) *Imbalance factor* (U0x), (b) τ (tau).

5 Conclusions

Dynamic load balancing in distributed systems based on application of the Extremal Optimization approach and global states monitoring has been discussed in this paper. The load balancing algorithm composed of iterative optimization phases based on the use of Extremal Optimization which define periodic real migration of the tasks proved to be an efficient and successful method for load balancing. The Extremal Optimization is executed in the background of application computations, in parallel with the application program.

The proposed algorithm including the Extremal Optimization has been assessed by experiments with simulated load balancing of distributed program graphs. In particular, the experiments compare the proposed load balancing method including the Extremal Optimization with an equivalent deterministic algorithm based on the similar theoretical foundations for load balancing. The comparison shows that the quality of load balancing with Extremal Optimization is in most cases better than that of the deterministic algorithm.

References

1. S. Boettcher, A.G. Percus: Extremal optimization: methods derived from coevolution. Proceedings of the Genetic and Evolutionary Computation Conference, (GECCO99), Morgan Kaufmann, San Francisco, CA (1999), 825–832.
2. R. Olejnik, I. Alshabani, B. Toursel, E. Laskowski, M. Tudruj, Load Balancing Metrics for the SOAJA Framework, Scalable Computing: Practice and Experience, 2009, Vol. 10, No. 4.
3. J. Borkowski, D. Kopański, E. Laskowski, R. Olejnik, M. Tudruj, A Distributed Program Global Execution Control Environment Applied to Load balancing, Scalable Computing: Practice and Experience, Vol. 13, No 3 (2012).
4. E. Laskowski, M. Tudruj, I. De Falco, U. Scafuri, E. Tarantino, R. Olejnik, Extremal Optimization Applied to Task Scheduling of Distributed Java Programs, EvoCOMNET 2011, LNCS 6625, Part II, 2011 Springer, pp. 61-70.
5. I. De Falco, E. Laskowski, R. Olejnik, U. Scafuri, E. Tarantino, M. Tudruj, Extremal Optimization Approach Applied to Initial Mapping of Distributed Java Programs, Euro-Par 2010, LNCS 6271, 2010, pp. 180-191.
6. K. Barker, N. Chrisochoides, An Evaluation of a Framework for the Dynamic Load Balancing of Highly Adaptive and Irregular Parallel Applications, Supercomputing 2003, Phoenix, ACM, 2003.
7. L.V. Kalé, S. Krishnan, CHARM++: A Portable Concurrent Object Oriented System Based on C++, Proc. of OOPSLA'93, ACM Press, Sept. 1993.
8. K.D. Devine, E.G. Boman, L.A. Riesen, U.V. Catalyurek, C. Chevalier, Getting Started with Zoltan: A Short Tutorial, Sandia National Labs Tech Report SAND2009-0578C, 2009.
9. C. Roig, A. Ripoll, F. Guirado, A New Task Graph Model for Mapping Message Passing Applications, IEEE Trans. on Parallel and Distributed Systems, Vol. 18 Issue 12, December 2007, pp. 1740–1753.
10. J.A. Hartigan, M.A. Wong, A K-Means clustering algorithm, Applied statistics, Vol. 28, pp. 100-108, 1979.
11. B. Zeigler, Hierarchical, modular discrete-event modelling in an object-oriented environment, Simulation 49 (5), 1987, pp. 219-230.